7

Hearing the Voice of the Customer

"Good morning, Maria. I'm Phil, the requirements analyst for the new employee information system we're going to build for you. Thanks for agreeing to be the product champion for this project. Your input will really help us a lot. So, can you tell me what you want?"

"Hmmm, what do I want," mused Maria. "I hardly know where to start. It should be a lot faster than the old system. And you know how the old system crashes if some employee has a really long name and we have to call the help desk and ask them to enter the name? The new system should take long names without crashing. Also, a new law says we can't use Social Security numbers for employee IDs anymore, so we'll have to change all the employee IDs when the new system goes in. The new IDs are going to be six-digit numbers. Oh, yes, it'd be great if I could get a report of how many hours of training each employee has had so far this year. And I also need to be able to change someone's name even if their marital status hasn't changed."

Phil dutifully wrote down everything Maria said, but his head was starting to spin. He wasn't sure what to do with all these bits of information, and he had no idea what to tell the developers. "Well," he thought, "if that's what Maria says she wants, I guess we'd better do it."

The heart of requirements engineering is *elicitation*, the process of identifying the needs and constraints of the various stakeholders for a software system. Elicitation focuses on discovering the user requirements, the middle level of the software requirements triad. (As described in Chapter 1, business requirements and functional requirements are the other two levels.) User requirements

encompass the tasks that users need to accomplish with the system and the users' expectations of performance, usability, and other quality attributes. This chapter addresses the general principles of effective requirements elicitation.

The analyst needs a structure to organize the array of input obtained from requirements elicitation. Simply asking the users, "What do you want?" generates a mass of random information that leaves the analyst floundering. "What do you need to do?" is a much better question. Chapter 8, "Understanding User Requirements," describes how techniques such as use cases and event-response tables provide helpful organizing structures for user requirements.

The product of requirements development is a common understanding among the project stakeholders of the needs that are being addressed. Once the developers understand the needs, they can explore alternative solutions to address those needs. Elicitation participants should resist the temptation to design the system until they understand the problem. Otherwise, they can expect to do considerable design rework as the requirements become better defined. Emphasizing user tasks rather than user interfaces and focusing on root needs more than on expressed desires help keep the team from being sidetracked by prematurely specifying design details.

Begin by planning the project's requirements elicitation activities. Even a simple plan of action increases the chance of success and sets realistic expectations for the stakeholders. Only by gaining explicit commitment on resources, schedule, and deliverables can you avoid having people pulled off elicitation to fix bugs or do other work. Your plan should address the following items:

- Elicitation objectives (such as validating market data, exploring use cases, or developing a detailed set of functional requirements for the system)
- Elicitation strategies and processes (for example, some combination of surveys, workshops, customer visits, individual interviews, and other techniques, possibly using different approaches for different stakeholder groups)
- Products of elicitation efforts (perhaps a list of use cases, a detailed SRS, an analysis of survey results, or performance and quality attribute specifications)
- Schedule and resource estimates (identify both development and customer participants in the various elicitation activities, along with estimates of the effort and calendar time required)
- Elicitation risks (identify factors that could impede your ability to complete the elicitation activities as intended, estimate the severity of each risk, and decide how you can mitigate or control it)

Requirements Elicitation

Requirements elicitation is perhaps the most difficult, most critical, most error-prone, and most communication-intensive aspect of software development. Elicitation can succeed only through a collaborative partnership between customers and the development team, as described in Chapter 2. The analyst must create an environment conducive to a thorough exploration of the product being specified. To facilitate clear communication, use the vocabulary of the application domain instead of forcing customers to understand computer jargon. Capture significant application domain terms in a glossary, rather than assuming that all participants share the same definitions. Customers should understand that a discussion about possible functionality is not a commitment to include it in the product. Brainstorming and imagining the possibilities is a separate matter from analyzing priorities, feasibility, and the constraining realities. The stakeholders must focus and prioritize the blue-sky wish list to avoid defining an enormous project that never delivers anything useful.

Skill in conducting elicitation discussions comes with experience and builds on training in interviewing, group facilitation, conflict resolution, and similar activities. As an analyst, you must probe beneath the surface of the requirements the customers present to understand their true needs. Simply asking "why" several times can move the discussion from a presented solution to a solid understanding of the problem that needs to be solved. Ask open-ended questions to help you understand the users' current business processes and to see how the new system could improve their performance. Inquire about possible variations in the user tasks that the users might encounter and ways that other users might work with the system. Imagine yourself learning the user's job, or actually do the job under the user's direction. What tasks would you need to perform? What questions would you have? Another approach is to play the role of an apprentice learning from the master user. The user you are interviewing then guides the discussion and describes what he or she views as the important topics for discussion.

Probe around the exceptions. What could prevent the user from successfully completing a task? How should the system respond to various error conditions? Ask questions that begin with "What else could…," "What happens when…," "Would you ever need to…," "Where do you get…," "Why do you (or don't you)…," and "Does anyone ever…" Document the source of each requirement so that you can obtain further clarification if needed and trace development activities back to specific customer origins.

When you're working on a replacement project for a legacy system, ask the users, "What three things annoy you the most about the existing system?"

This question helps get to the bottom of why a system is being replaced. It also surfaces expectations that the users hold for the follow-on system. As with any improvement activity, dissatisfaction with the current situation provides excellent fodder for the new and improved future state.

Try to bring to light any assumptions the customers might hold and to resolve conflicting assumptions. Read between the lines to identify features or characteristics the customers expect to be included without their having explicitly said so. Gause and Weinberg (1989) suggest using *context-free questions*, high-level and open-ended questions that elicit information about global characteristics of both the business problem and the potential solution. The customer's response to questions such as "What kind of precision is required in the product?" or "Can you help me understand why you don't agree with Miguel's reply?" can lead to insights that questions with standard yes/no or A/B/C answers do not.

Rather than simply transcribing what customers say, a creative analyst suggests ideas and alternatives to users during elicitation. Sometimes users don't realize the capabilities that developers can provide and they get excited when you suggest functionality that will make the system especially useful. When users truly can't express what they need, perhaps you can watch them work and suggest ways to automate portions of the job. Analysts can think outside the box that limits the creativity of people who are too close to the problem domain. Look for opportunities to reuse functionality that's already available in another system.

Interviews with individuals or groups of potential users are a traditional source of requirements input for both commercial products and information systems. (For guidance on how to conduct user interviews, see Beyer and Holtzblatt [1998], Wood and Silver [1995], and McGraw and Harbison [1997].) Engaging users in the elicitation process is a way to gain support and buy-in for the project. Try to understand the thought processes that led the users to present the requirements they state. Walk through the processes that users follow to make decisions about their work and extract the underlying logic. Flowcharts and decision trees are useful ways to depict these logical decision paths. Make sure that everyone understands why the system *must* perform certain functions. Proposed requirements sometimes reflect obsolete or ineffective business processes that should not be incorporated into a new system.

After each interview, document the items that the group discussed and ask the interviewees to review the list and make corrections. Early review is essential to successful requirements development because only those people who supplied the requirements can judge whether they were captured accurately. Use further discussions to resolve any inconsistencies and to fill in any blanks.

Elicitation Workshops

Requirements analysts frequently facilitate requirements elicitation workshops. Facilitated, collaborative group workshops are a highly effective technique for linking users and developers (Keil and Carmel 1995). The facilitator plays a critical role in planning the workshop, selecting participants, and guiding the participants to a successful outcome. When a team is getting started with new approaches to requirements elicitation, have an outside facilitator lead the initial workshops. This way the analyst can devote his full attention to the discussion. A scribe assists the facilitator by capturing the points that come up during the discussion.

According to one authority, "Facilitation is the art of leading people through processes toward agreed-upon objectives in a manner that encourages participation, ownership, and productivity from all involved" (Sibbet 1994). A definitive resource on facilitating requirements elicitation workshops is Ellen Gottesdiener's *Requirements by Collaboration* (2002). Gottesdiener describes a wealth of techniques and tools for workshop facilitation. Following are a few tips for conducting effective elicitation sessions.

Establish ground rules. The participants should agree on some basic operating principles for their workshops (Gottesdiener 2002). Examples include the following:

- Starting and ending meetings on time
- Returning from breaks promptly
- Holding only one conversation at a time
- Expecting everyone to contribute
- Focusing comments and criticisms on issues, not on individuals

Stay in scope. Use the vision and scope document to confirm whether proposed user requirements lie within the current project scope. Keep each workshop focused on the right level of abstraction for that day's objectives. Groups easily dive into distracting detail during requirements discussions. Those discussions consume time that the group should spend initially on developing a higher-level understanding of user requirements; the details will come later. The facilitator will have to reel in the elicitation participants periodically to keep them on topic.

> **Trap** Avoid drilling down into excessive requirements detail prematurely. Recording great detail about what people already understand doesn't reduce the risks due to uncertainty in the requirements.

It's easy for users to begin itemizing the precise layout of items in a report or a dialog box before the team even agrees on the pertinent user task. Recording these details as requirements places unnecessary constraints on the subsequent design process. Detailed user interface design comes later, although preliminary screen sketches can be helpful at any point to illustrate how you might implement the requirements. Early feasibility exploration, which requires some amount of design, is a valuable risk-reduction technique.

Use parking lots to capture items for later consideration. An array of random but important information will surface in an elicitation workshop: quality attributes, business rules, user interface ideas, constraints, and more. Organize this information on flipcharts—parking lots—so that you don't lose it and to demonstrate respect for the participant who brought it up. Don't be distracted into discussing off-track details unless they turn out to be showstopper issues, such as a vital business rule that restricts the way a use case can work.

Timebox discussions. The facilitator might allocate a fixed period of time to each discussion topic, say, 30 minutes per use case during initial use case explorations. The discussion might need to be completed later, but timeboxing helps avoid the trap of spending far more time than intended on the first topic and neglecting the other planned topics entirely.

Keep the team small and include the right participants. Small groups can work much faster than larger teams. Elicitation workshops with more than five or six active participants can become mired in side trips down "rat holes," concurrent conversations, and bickering. Consider running multiple workshops in parallel to explore the requirements of different user classes. Workshop participants should include the product champion and other user representatives, perhaps a subject matter expert, a requirements analyst, and a developer. Knowledge, experience, and authority to make decisions are qualifications for participating in elicitation workshops.

> **Trap** Watch out for off-topic discussions, such as design explorations, during elicitation sessions.

Keep everyone engaged. Sometimes participants will stop contributing to the discussion. These people might be frustrated because they see that the system is an accident waiting to happen. Perhaps their input isn't being taken seriously because other participants don't find their concerns interesting or don't want to disrupt the work that the group has completed so far. Perhaps the stakeholder who has withdrawn has a submissive personality and is deferring to more aggressive participants or a dominating analyst. The facilitator must read the body language, understand why someone has tuned out of the process, and try to bring the person back. That individual might hold an insightful perspective that could make an important contribution.

Too Many Cooks

Requirements elicitation workshops that involve too many participants can slow to a contentious crawl. My colleague Debbie was frustrated at the sluggish progress of the first use-case workshop she facilitated for a Web development project. The 12 participants held extended discussions of unnecessary details and couldn't agree on how each use case ought to work. The team's progress accelerated nicely when Debbie reduced the number of participants to six who represented the roles of analyst, customer, system architect, developer, and visual designer. The workshop lost some input by using the smaller team but the rate of progress more than compensated for that loss. The workshop participants should exchange information off-line with colleagues who don't attend and then bring the collected input to the workshops.

Classifying Customer Input

Don't expect your customers to present a succinct, complete, and well-organized list of their needs. Analysts must classify the myriad bits of requirements

information they hear into various categories so that they can document and use it appropriately. Figure 7-1 illustrates nine such requirement categories.

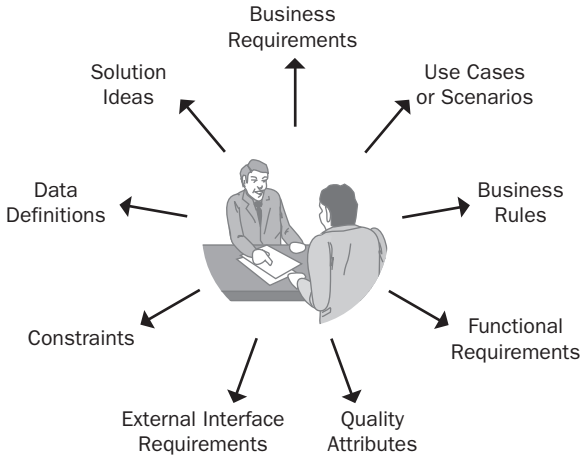

Figure 7-1 Classifying the voice of the customer.

Information that doesn't fit into one of these buckets might be one of the following:

- A requirement not related to the software development, such as the need to train users on the new system
- A project constraint, such as a cost or schedule restriction (as opposed to the design or implementation constraints described in this chapter)
- An assumption
- A data requirement, which can often be associated with some system functionality (you store data in a computer only so that you can get it out again later)
- Additional information of a historical, context-setting, or descriptive nature

The following discussion suggests some phrases to listen for that will help you in this classification process.

Business requirements. Anything that describes the financial, marketplace, or other business benefit that either customers or the developing organization

wish to gain from the product is a business requirement. Listen for statements about the value that buyers or users of the software will receive, such as these:

- "Increase market share by X%."
- "Save $Y per year on electricity now wasted by inefficient units."
- "Save $Z per year in maintenance costs that are consumed by legacy system W."

Use cases or scenarios. General statements of user goals or business tasks that users need to perform are use cases; a single specific path through a use case is a usage scenario. Work with the customers to generalize specific scenarios into more abstract use cases. You can often glean use cases by asking users to describe their business workflow. Another way to discover use cases is to ask users to state the goals they have in mind when they sit down to work with the system. A user who says, "I need to <do something>" is probably describing a use case, as in the following examples:

- "I need to print a mailing label for a package."
- "I need to manage a queue of chemical samples waiting to be analyzed."
- "I need to calibrate the pump controller."

Business rules. When a customer says that only certain user classes can perform an activity under specific conditions, he might be describing a business rule. In the case of the Chemical Tracking System, such a business rule might be, "A chemist may order a chemical on the Level 1 hazard list only if his hazardous-chemical training is current." You might derive some software functional requirements to enforce the rules, such as making the training record database accessible to the Chemical Tracking System. As stated, though, business rules are not functional requirements. Following are some other phrases that suggest the user is describing a business rule:

- "Must comply with <some law or corporate policy>"
- "Must conform to <some standard>"
- "If <some condition is true>, then <something happens>"
- "Must be calculated according to <some formula>"

> **More Info** See Chapter 9, "Playing by the Rules," for more examples of business rules.

Functional requirements. Functional requirements describe the observable behaviors the system will exhibit under certain conditions and the actions the system will let users take. Functional requirements derived from system requirements, user requirements, business rules, and other sources make up the bulk of the SRS. Here are some examples of functional requirements as you might hear them from users:

- "If the pressure exceeds 40.0 psi, the high pressure warning light should come on."
- "The user must be able to sort the project list in forward and reverse alphabetical order."
- "The system sends an e-mail to the Idea Coordinator whenever someone submits a new idea."

These statements illustrate how users typically present functional requirements, but they don't represent good ways to write functional requirements in an SRS. In the first case, we would replace *should* with *shall* to make it clear that illuminating the warning light is essential. The second example is a requirement of the user, not of the system. The requirement of the system is to permit the user to do the sorting.

> **More Info** Chapter 10, "Documenting the Requirements," contains more guidance for writing good functional requirements.

Quality attributes. Statements that indicate how well the system performs some behavior or lets the user take some action are quality attributes. Listen for words that describe desirable system characteristics: fast, easy, intuitive, user-friendly, robust, reliable, secure, and efficient. You'll have to work with the users to understand precisely what they mean by these ambiguous and subjec-

tive terms and write clear, verifiable quality goals, as described in Chapter 12, "Beyond Functionality: Software Quality Attributes."

External interface requirements. Requirements in this class describe the connections between your system and the rest of the universe. The SRS should include sections for interfaces to users, hardware, and other software systems. Phrases that indicate that the customer is describing an external interface requirement include the following:

- "Must read signals from <some device>"
- "Must send messages to <some other system>"
- "Must be able to read (or write) files in <some format>"
- "Must control <some piece of hardware>"
- "User interface elements must conform to <some UI style standard>"

Constraints. Design and implementation constraints legitimately restrict the options available to the developer. Devices with embedded software often must respect physical constraints such as size, weight, and interface connections. Record the rationale behind each constraint so that all project participants know where it came from and respect its validity. Is it truly a restrictive limitation, as when a device must fit into an existing space? Or is it a desirable goal, such as a portable computer that weighs as little as possible?

Unnecessary constraints inhibit creating the best solution. Constraints also reduce your ability to use commercially available components as part of the solution. A constraint that specifies that a particular technology be used poses the risk of making a requirement obsolete or unattainable because of changes in the available technologies. Certain constraints can help achieve quality attribute goals. An example is to improve portability by using only the standard commands of a programming language, not permitting vendor-specific extensions. The following are examples of constraints that a customer might present:

- "Files submitted electronically may not exceed 10 MB in size."
- "The browser must use 128-bit encryption for all secure transactions."
- "The database must use the Framalam 10.2 run-time engine."

Other phrases that indicate the customer is describing a design or implementation constraint include these:

- "Must be written in <a specific programming language>"
- "Can't require more than <some amount of memory>"
- "Must operate identically to (or be consistent with) <some other system>"
- "Must use <a specific user interface control>"

As with functional requirements, the analyst shouldn't simply transcribe the user's statement of a constraint into the SRS. Weak words such as *identically* and *consistent* need to be clarified and the real constraint stated precisely enough for developers to act on the information. Ask why the constraint exists, verify its validity, and record the rationale for including the constraint as a requirement.

Data definitions. Whenever customers describe the format, data type, allowed values, or default value for a data item or the composition of a complex business data structure, they're presenting a data definition. "The ZIP code consists of five digits, followed by an optional hyphen and an optional four digits that default to 0000" is a data definition. Collect these in a *data dictionary*, a master reference that the team can use throughout the product's development and maintenance.

> **More Info** See Chapter 10 for more information on data dictionaries.

Data definitions sometimes lead to functional requirements that the user community did not request directly. What happens when a six-digit order number rolls over from 999,999? Developers need to know how the system will handle such data issues. Deferring data-related problems just makes them harder to solve in the future (remember Y2K?).

Solution ideas. Much of what users present as requirements fits in the category of solution ideas. Someone who describes a specific way to interact with the system to perform some action is presenting a suggested solution. The ana-

lyst needs to probe below the surface of a solution idea to get to the real requirement. For instance, functional requirements that deal with passwords are just one of several possible solutions for a security requirement.

Suppose a user says, "Then I select the state where I want to send the package from a drop-down list." The phrase *from a drop-down list* indicates that this is a solution idea. The prudent analyst will ask, "Why from a drop-down list?" If the user replies, "That just seemed like a good way to do it," the real requirement is something like, "The system shall permit the user to specify the state where he wants to send the package." However, maybe the user says, "I suggested a drop-down list because we do the same thing in several other places and I want it to be consistent. Also, it prevents the user from entering invalid data, and I thought we might be able to reuse some code." These are fine reasons to specify a specific solution. Recognize, though, that embedding a solution idea in a requirement imposes a design constraint on that requirement. It limits the requirement to being implemented in only one way. This isn't necessarily wrong or bad; just make sure the constraint is there for a good reason.

Some Cautions About Elicitation

Trying to amalgamate requirements input from dozens of users is difficult without using a structured organizing scheme, such as use cases. Collecting input from too few representatives or hearing the voice only of the loudest, most opinionated customer is also a problem. It can lead to overlooking requirements that are important to certain user classes or to including requirements that don't represent the needs of a majority of the users. The best balance involves a few product champions who have authority to speak for their respective user classes, with each champion backed up by several other representatives from that same user class.

During requirements elicitation, you might find that the project scope is improperly defined, being either too large or too small (Christel and Kang 1992). If the scope is too large, you'll collect more requirements than are needed to deliver adequate business and customer value and the elicitation process will drag on. If the project is scoped too small, customers will present needs that are clearly important yet just as clearly lie beyond the limited scope currently established for the project. The present scope could be too small to yield a satisfactory product. Eliciting user requirements therefore can lead to modifying the product vision or the project scope.

It's often stated that requirements are about *what* the system has to do, whereas *how* the solution will be implemented is the realm of design. Although attractively concise, this is an oversimplification. Requirements elicitation should indeed focus on the *what*, but there's a gray area—not a sharp line—between analysis and design. Hypothetical *how*s help to clarify and refine the understanding of what users need. Analysis models, screen sketches, and prototypes help to make the needs expressed during requirements elicitation more tangible and to reveal errors and omissions. Regard the models and screens generated during requirements development as conceptual suggestions to facilitate effective communication, not as constraints on the options available to the designer. Make it clear to users that these screens and prototypes are illustrative only, not necessarily the final design solution.

The need to do exploratory research sometimes throws a monkey wrench into the works. An idea or a suggestion arises, but extensive research is required to assess whether it should even be considered for possible incorporation into the product. Treat these explorations of feasibility or value as project tasks in their own right, with objectives, goals, and requirements of their own. Prototyping is one way to explore such issues. If your project requires extensive research, use an incremental development approach to explore the requirements in small, low-risk portions.

Finding Missing Requirements

Missing requirements constitute the most common type of requirement defect (Jones 1997). They're hard to spot during reviews because they're invisible! The following techniques will help you detect previously undiscovered requirements.

> **Trap** Watch out for the dreaded *analysis paralysis*, spending too much time on requirements elicitation in an attempt to avoid missing any requirements. You'll never discover them all up front.

- Decompose high-level requirements into enough detail to reveal exactly what is being requested. A vague, high-level requirement that leaves much to the reader's interpretation will lead to a gap between what the requester has in mind and what the developer

builds. Imprecise, fuzzy terms to avoid include *support*, *enable*, *permit*, *process*, and *manage*.

- Make sure that all user classes have provided input. Make sure that each use case has at least one identified actor.
- Trace system requirements, use cases, event-response lists, and business rules into their detailed functional requirements to make sure that the analyst derived all the necessary functionality.
- Check boundary values for missing requirements. Suppose that one requirement states, "If the price of the order is less than $100, the shipping charge is $5.95" and another says, "If the price of the order is more than $100, the shipping charge is 5 percent of the total order price." But what's the shipping charge for an order with a price of exactly $100? It's not specified, so a requirement is missing.
- Represent requirements information in multiple ways. It's difficult to read a mass of text and notice that something isn't there. An analysis model visually represents requirements at a high level of abstraction—the forest, not the trees. You might study a model and realize that there should be an arrow from one box to another; that missing arrow represents a missing requirement. This kind of error is much easier to spot in a picture than in a long list of textual requirements that all blur together. Analysis models are described in Chapter 11, "A Picture Is Worth 1024 Words."
- Sets of requirements with complex Boolean logic (ANDs, ORs, and NOTs) often are incomplete. If a combination of logical conditions has no corresponding functional requirement, the developer has to deduce what the system should do or chase down an answer. Represent complex logic using decision tables or decision trees to make sure you've covered all the possible situations, as described in Chapter 11.

A rigorous way to search for missing requirements is to create a CRUD matrix. *CRUD* stands for *Create*, *Read*, *Update*, and *Delete*. A CRUD matrix correlates system actions with data entities (individual data items or aggregates of data items) to make sure that you know where and how each data item is created, read, updated, and deleted. Some people add an *L* to the matrix to indicate that the data item appears as a *List* selection (Ferdinandi 2002). Depending on the requirements analysis approaches you are using, you can examine various types of correlations, including the following:

- Data entities and system events (Robertson and Robertson 1999)
- Data entities and user tasks or use cases (Lauesen 2002)
- Object classes and system events (Ferdinandi 2002)
- Object classes and use cases (Armour and Miller 2001)

Figure 7-2 illustrates an entity/use case CRUDL matrix for a portion of the Chemical Tracking System. Each cell indicates how the use case in the leftmost column uses each data entity shown in the other columns. The use case can **C**reate, **R**ead, **U**pdate, **D**elete, or **L**ist the entity. After creating a CRUDL matrix, see whether any of these five letters do not appear in any of the cells in a column. If a business object is updated but never created, where does it come from? Notice that none of the cells under the column labeled Requester (the person who places an order for a chemical) contains a *D*. That is, none of the use cases in Figure 7-2 can delete a Requester from the list of people who have ordered chemicals. There are three possible interpretations:

1. Deleting a Requester is not an expected function of the Chemical Tracking System.
2. We are missing a use case that deletes a Requester.
3. The "Edit Requesters" use case is incorrect. It's supposed to permit the user to delete a Requester, but that functionality is missing from the use case at present.

We don't know which interpretation is correct, but the CRUDL analysis is a powerful way to detect missing requirements.

Entity / Use Case	Order	Chemical	Requester	Vendor Catalog
Place Order	C	R	R	R, L
Change Order	U, D		R	R, L
Manage Chemical Inventory		C, U, D		
Report on Orders	R	R, L	R, L	
Edit Requesters			C, U, L	

Figure 7-2 Sample CRUDL matrix for the Chemical Tracking System.

How Do You Know When You're Done?

No simple signal will indicate when you've completed requirements elicitation. As people muse in the shower each morning and talk with their colleagues, they'll generate ideas for additional requirements. You'll never be completely done, but the following cues suggest that you're reaching the point of diminishing returns on requirements elicitation:

- If the users can't think of any more use cases, perhaps you're done. Users tend to identify use cases in sequence of decreasing importance.
- If users propose new use cases but you've already derived the associated functional requirements from other use cases, perhaps you're done. These "new" use cases might really be alternative courses for other use cases that you've already captured.
- If users repeat issues that they already covered in previous discussions, perhaps you're done.
- If suggested new features, user requirements, or functional requirements are all out of scope, perhaps you're done.
- If proposed new requirements are all low priority, perhaps you're done.
- If the users are proposing capabilities that might be included "sometime in the lifetime of the product" rather than "in the specific product we're talking about right now," perhaps you're done, at least with the requirements for the next release.

Another way to determine whether you're done is to create a checklist of common functional areas to consider for your projects. Examples include error logging, backup and restore, access security, reporting, printing, preview capabilities, and configuring user preferences. Periodically compare this list with the functions you have already specified. If you don't find gaps, perhaps you're done.

Despite your best efforts to discover *all* the requirements, you won't, so expect to make changes as construction proceeds. Remember, your goal is to make the requirements good enough to let construction proceed at an acceptable level of risk.

Next Steps

- Think about missing requirements that were discovered late on your last project. Why were they overlooked during elicitation? How could you have discovered each of these requirements earlier?
- Select a portion of any documented voice-of-the-customer input on your project or a section from the SRS. Classify every item in that requirements fragment into the categories shown in Figure 7-1 (on page 120): business requirements, use cases or scenarios, business rules, functional requirements, quality attributes, external interface requirements, constraints, data definitions, and solution ideas. If you discover items that are classified incorrectly, move them to the correct place in the requirements documentation.
- List the requirements elicitation methods used on your current project. Which ones worked well? Why? Which ones did not work so well? Why not? Identify elicitation techniques that you think would work better and decide how you'd apply them next time. Identify any barriers you might encounter to making those techniques work and brainstorm ways to overcome those barriers.